PART 5 OF 8

Python Self-Learning Booklet 5: Functions

Welcome to Part 5! In this session you will learn how to create and use **functions** in Python – one of the most powerful tools in the language. Functions let you name and reuse blocks of code, keeping your programs clean and organized.

6	6	5
Learning Goals	**Core Concepts**	**Practice Tasks**
Core skills to master in this session	From basics to *args and **kwargs	Hands-on exercises to reinforce your learning

SESSION OVERVIEW

What You Will Learn

By the end of this session, you should be able to do all of the following:

1 Define and call functions

Use `def` to create a function and call it to make it run.

2 Pass data using arguments

Use positional, keyword, default, and optional arguments.

3 Return values

Send results back with `return` and capture them in variables.

4 Work with lists and flexible inputs

Pass lists safely, and use `*args` and `**kwargs` for variable-length inputs.

CONCEPT 1

What Is a Function?

A **function** is a *named block of code* that performs one task. Instead of writing the same logic repeatedly, you define it once and call it whenever you need it.

Why Use Functions?

- **Reuse code** instead of rewriting it
- **Organize programs** into small, clear parts
- **Improve readability** with meaningful names

You already know built-in functions: `len()`, `sorted()`, `range()`

Your First Function

```
def greet_user():
    print("Hello!")

greet_user()
```

- `def greet_user():` defines the function
- The indented lines are the **function body**
- `greet_user()` calls (runs) it
- Without the call, nothing happens

CONCEPT 2

Parameters and Arguments

Functions can accept **input values**. There is an important distinction between two related terms:

Parameter

The variable name *inside the function definition*. It acts as a placeholder.

```
def greet_user(name):
    ...
```

Here, `name` is the parameter.

Argument

The *actual value* you pass in when calling the function.

```
greet_user("Lina")
greet_user("Maxi")
```

Here, `"Lina"` and `"Maxi"` are arguments.

Multiple Parameters

```
def describe_pet(animal, name):
    print("I have a", animal, "named", name)

describe_pet("dog", "Max")
describe_pet("Max", "dog")  # swapped — changes the meaning!
```

⚠️ **Order matters!** With positional arguments, Python matches values left to right. Swapping the order produces different (and wrong) output. You will fix this with keyword arguments next.

CONCEPT 3 – PART A

Positional and Keyword Arguments

Positional Arguments

The most common style – values matched by their position in the call.

```
def describe_pet(animal, name):
    print("I have a", animal,
          "named", name)

describe_pet("dog", "Max")
```

Keyword Arguments

Specify parameter names explicitly – order no longer matters.

```
describe_pet(animal="dog",
             name="Max")

describe_pet(name="Max",
             animal="dog")
```

Both calls produce **identical output** because Python matches by name.

CONCEPT 3 – PART B

Default Values

You can give a parameter a **default value**. If the caller does not provide that argument, Python uses the default automatically.

```
def describe_pet(name, animal="dog"):
    print("I have a", animal, "named", name)

describe_pet(name="Max", animal="dog")  # explicit
describe_pet(name="Max")                # uses default: animal="dog"
describe_pet(name="Max", animal="cat") # overrides default
```

📌 **Important rule:** Default parameters must always come **after** required parameters in the function definition. Writing `def f(animal="dog", name)` would cause a `SyntaxError`.

Avoiding Argument Errors

If a parameter has no default value, you *must* provide it – otherwise Python raises a `TypeError`.

```
def greet_user(name):
    print("Hello,", name)

# greet_user()  ← TypeError: missing 1 required positional argument
```

CONCEPT 3 – PART C

Optional Arguments with "" or None

Sometimes an input is truly optional – you want the function to behave differently depending on whether it was provided. The standard pattern is to default to `""` or `None` and then check with `if`.

Using an empty string ""

```
def print_full_name(first, last,
                    middle=""):
    if middle:
        print(first, middle,
              ",", last)
    else:
        print(first, ",", last)

print_full_name("John", "Doe")
print_full_name("John", "Doe",
                "Lee")
```

Using None

```
def print_full_name(first, last,
                    middle=None):
    if middle:
        print(first, middle,
              ",", last)
    else:
        print(first, ",", last)

print_full_name("John", "Doe")
print_full_name("John", "Doe",
                "Lee")
```

`None` is preferred – it clearly signals "no value provided".

CONCEPT 4 – PART A

Returning Values

So far, functions have only `print()`-ed results. But often you want a function to **send a value back** so you can store it and use it elsewhere. That is what `return` does.

1

Print only

```
def add(a, b):
    print(a + b)

add(1, 2)
# Output shown, but
# value is lost
```

2

Return

```
def add(a, b):
    return a + b
```

The result is sent back to the caller.

3

Capture & reuse

```
c = add(1, 2)
print(c)
# 3
```

Store the result in a variable for later use.

CONCEPT 4 – PART B

Returning Text, Dictionaries, and Conditional Values

Functions can return any Python value – strings, lists, or dictionaries. They can also decide *what* to return using `if/elif/else`.

Returning a formatted string

```
def get_full_name(first, last):
    full = first + " " + last
    return full.title()

name = get_full_name("jimi",
                     "hendrix")
print(name)  # Jimi Hendrix
```

Returning a dictionary

```
def build_person(first, last, age):
    person = {"first": first,
              "last": last}
    person["age"] = age
    return person

someone = build_person("jimi",
                       "hendrix", 27)
print(someone["first"])  # jimi
```

Conditional return

```
def pos_or_neg(number):
    if number < 0:
        return "Negative"
    elif number > 0:
        return "Positive"

r = pos_or_neg(3)
print(r)  # Positive
```

Return values in a loop

```
def square(n):
    return n ** 2

numbers = [1, 2, 3, 4]
results = []
for n in numbers:
    results.append(square(n))

print(results)  # [1, 4, 9, 16]
```

CONCEPT 5

Passing Lists to Functions (and Copying Them)

You can pass a list into a function just like any other value. However, be careful: if the function modifies the list, those changes affect the **original** list too – unless you pass a copy.

```
def move(src):
    dst = []
    while src:
        dst.append(src.pop())
    return dst

old = ['a', 'b']
new = move(old[:])   # old[:] creates a copy — old stays unchanged!
print(new)          # ['b', 'a']
print(old)          # ['a', 'b']
```

What `move()` does

It removes items from `src` one by one using `.pop()` and appends them to `dst` – reversing the order in the process.

Why use `old[:]`?

`old[:]` creates a **shallow copy** of the list. Passing the copy means `old` is never touched by the function's `.pop()` calls.

Rule of thumb

If you do *not* want a function to change the original list, always pass a copy with `my_list[:]`.

CONCEPT 6 – PART A

Arbitrary Arguments with *args

Sometimes you do not know how many arguments a function will receive. Python lets you handle this with *args, which collects any extra positional arguments into a **tuple**.

Basic *args

```
def pizza(*args):
    print('Pizza with:', args)

pizza('cheese')
# Pizza with: ('cheese',)

pizza('mushroom', 'onion')
# Pizza with: ('mushroom', 'onion')
```

Mixing with normal parameters

Regular parameters must come **first**, before *args.

```
def pizza(size, *t):
    print(size, 'inch:', t)

pizza(12, 'cheese', 'olives')
# 12 inch: ('cheese', 'olives')
```

Tuple reminder

```
t = ("cheese", "mushroom")  # immutable
l = ["cheese", "mushroom"]  # mutable
```

A tuple uses () and cannot be changed after creation.

CONCEPT 6 – PART B

Arbitrary Keyword Arguments with **kwargs

kwargs works like *args but for **keyword arguments. It collects any extra key=value pairs into a **dictionary**.

```
def car(make, model, **x):
    x['make'] = make
    x['model'] = model
    return x

car1 = car('BMW', 'X3', color='red')
print(car1)
# {'color': 'red', 'make': 'BMW', 'model': 'X3'}
```

*args → Tuple

Collects extra **positional** arguments.
Accessed by index: args[0], args[1], ...

**kwargs → Dictionary

Collects extra **keyword** arguments.
Accessed by key: kwargs["color"].

Parameter Order

Always order as: regular → *args → **kwargs in the function definition.

STEP-BY-STEP EXAMPLE

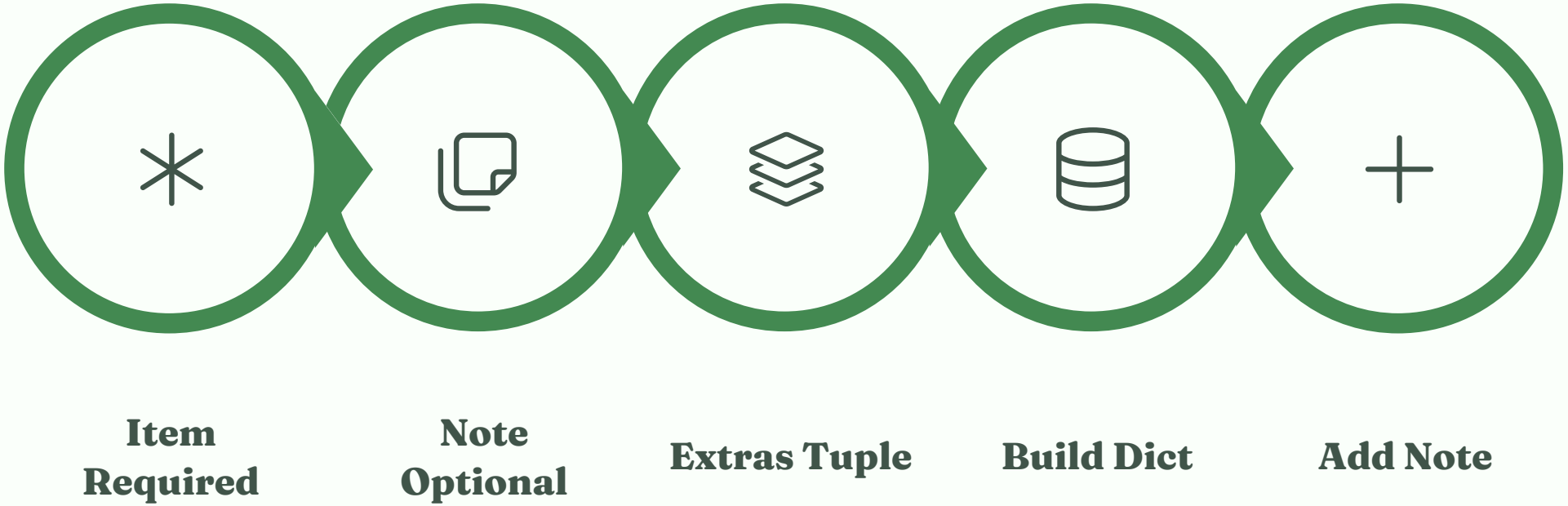
Building an Order: Putting It All Together

This extended example combines default values, optional `None`, returning a dictionary, and `*args`. Read through it carefully before trying to run it.

```
def build_order(item, size="medium", note=None, *extras):
    order = {
        "item": item,
        "size": size,
        "extras": list(extras)
    }

    if note:
        order["note"] = note

    return order


order1 = build_order("sandwich")
order2 = build_order("sandwich", "large", None, "cheese", "tomato")
order3 = build_order("sandwich", note="no onions", size="small")

print(order1)
print(order2)
print(order3)
```

STEP-BY-STEP EXAMPLE

How `build_order` Works — Step by Step



Follow these six steps each time you read a new function. Understanding *what each parameter does* and *what is returned* is the key skill for working with functions.

WATCH OUT

Common Mistakes to Avoid

1

Defining but never calling

A function does nothing until you call it. Writing `def greet():` alone produces no output.

2

Forgetting indentation

Python requires consistent indentation inside a function body. Missing or inconsistent spaces cause `IndentationError`.

3

Mixing up parameter vs. argument

Parameter lives in `def`. Argument is the value you pass in the call. Confusing them makes debugging harder.

4

Wrong positional argument order

`describe_pet("Max", "dog")` vs `describe_pet("dog", "Max")` – swapping produces confusing, incorrect output.

5

Accidentally modifying a list

If a function uses `.pop()` or similar, the original list is changed. Pass a copy with `my_list[:]` to prevent this.

PRACTICE

Practice Tasks

Complete these tasks on your own. Each one targets a specific skill from this session. Try to write and run each function before checking any solutions.

01

Task 1 — Basic function call

Write a function `greet()` that prints a short greeting. Call it **three times**.

02

Task 2 — Default argument

Write `describe_city(city, country="Unknown")` that prints: *"<city> is in <country>."* Call it once with only a city, and once with both.

03

Task 3 — Return value

Write `add(a, b)` that **returns** the sum. Store the result in a variable and print it.

04

Task 4 — Return a dictionary

Write `make_profile(first, last, age)` that returns a dictionary with keys `"first"`, `"last"`, `"age"`.

05

Task 5 — `*args`

Write `sandwich(*args)` that prints all provided toppings. It should work with 0, 1, or many toppings.

EXPLORATION

Exploration Questions

These questions do not have a single right answer – they are designed to deepen your understanding. Try them out in Python before forming your conclusion.

Question 1 — List mutation

In the list-moving example, what changes if you call `move(old)` instead of `move(old[:])`? Why does the original list end up empty?

Question 2 — Empty string vs. `None`

In the `build_order()` example, what happens if you pass `note=""` (empty string) instead of `None`? Does the `if note:` check behave the same way?

Question 3 — Extended `**kwargs`

Modify the `car()` function so it accepts `make` and `model`, plus extras like `year=2020` and `electric=True`. What does the returned dictionary look like?

SUMMARY

Session 5 Summary: Functions

Here is a quick reference of everything covered in this session. Use this as a checklist before moving on to Part 6.

Define & Call

Use `def` to create a function. Call it by name with `()`. Nothing runs without a call.

Parameters & Arguments

Parameter = placeholder in `def`. Argument = actual value in the call.

Argument Types

Positional, keyword, default values, and optional (`""` or `None`) give you flexible control.

Return Values

`return` sends a value back. Capture it in a variable to store and reuse it.

Lists & Copies

Pass lists with `my_list[:]` when you want the original to stay unchanged.

*args / **kwargs

`*args` → tuple of extra positional inputs. `**kwargs` → dictionary of extra keyword inputs.

✅ **Ready for Part 6?** Make sure you can complete all five practice tasks confidently before moving on. Functions are the foundation for everything that follows!